MULTICS SECURITY KERNEL CERTIFICATION PLAN

Honeywell Information Systems, Inc.
Federal Systems Operations
7900 Westpark Drive
McLean, VA 22101

and

Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, CA 94025

July 1976

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

"This technical report has been reviewed and is approved for publication."

William R. Price

WILLIAM R. PRICE, Capt, USAF
Techniques Engineering Division

Donald P. Eriksen

DONALD P. ERIKSEN
Techniques Engineering Division

FOR THE COMMANDER

Stanley P. Dereska

STANLEY P. DERESKA, Colonel, USAF
Deputy Director, Computer Systems Engineering
Deputy for Command and Management Systems

SECURITY CLASSIFICATION OF THIS PAGE *(When Date Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ESD-TR-76-297
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE *(and Subtitle)*: MULTICS SECURITY KERNEL CERTIFICATION PLAN

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s):

8. CONTRACT OR GRANT NUMBER(s): F19628-74-C-0193

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Honeywell Info. Systems, Inc.
Federal Systems Operations
7900 Westpark Drive
McLean, VA 22101

Stanford Res. Institute
333 Ravenswood Ave.
Menlo Park, CA 94025

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: CDRL Item A020

11. CONTROLLING OFFICE NAME AND ADDRESS:
Deputy for Command and Management Systems
Electronic Systems Division
Hanscom AFB, MA 01731

12. REPORT DATE: July 1976

13. NUMBER OF PAGES: 43

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*:

15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*:
Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*:

18. SUPPLEMENTARY NOTES:

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*:
Multilevel security certification
security kernel
top-level specifications
Specification and Assertion Language
O-Functions
V-Functions
Correspondence Proof
Correctness Proof

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*:

This report describes the methodology for formal certification approach of a security kernel implementation with respect to the top-level specifications of that kernel. An illustration of the proofs of correspondence between the kernel specifications and the desired multilevel properties (the security model) is included in the report. This methodology developed by Stanford Research Institute employs a formal hierarchical decomposition of the design,
(Continued on reverse side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Dete Entered)*

20. ABSTRACT

with formally stated specifications for each desired property. The report describes this methodology and its application to the task of developing the certifiable security kernels for Multics and the Secure Front-End Processor (SFEP).

PREFACE

This report presents a plan to certify the security kernels of Multics and its Secure Front-End Processor (SFEP) for Project Guardian. This document was prepared by F. J. Feiertag, K. N. Levitt, P. G. Neumann, and L. Robinson of Stanford Research Institute as a subcontractor to Honeywell in this program.

Because of funding limitations, the Air Force terminated the effort which this document describes before the effort reached its logical conclusion. This report is incomplete but was published in the interest of capturing and disseminating the computer security technology that was available when the effort was terminated. Air Force technical comments are included as an appendix to identify unresolved technical issues at the time of the report.

This report was to describe a methodology for demonstrating that a specific security kernel implementation effectively provided the security controls which a mathematical model has shown to be sufficient to comply with the Department of Defense Information Security Program. However, the report only describes the methodology for demonstrating the correspondence between the model and a high level specification of a security kernel. The effort was terminated before developing techniques to deal with lower level representations of the security kernel.

TABLE OF CONTENTS

I. BACKGROUND

SUMMARY

This document outlines the approach considered feasible for the development of a security kernel for Multics whose security properties can be formally verified with respect to the top-level specifications of the kernel. An illustration of the proofs of correspondence between the kernel specifications and the desired multilevel properties is given in Appendix A. It is also shown how these proofs may be carried out automatically. The approach given here is also applicable to proofs of program correctness, i.e., consistency between the specifications and the programs implementing those specifications. However, proofs of program correctness are not considered here.

The approach relies heavily on the use of a formal methodology for the design, implementation, and proof of computer systems. This methodology has been developed at Stanford Research Institute (SRI) and is being applied to the design of several systems, including a provably secure operating system (PSOS) and several user environments intended to be implemented on it, an ultra-reliable computing system with software-implemented fault tolerance (SIFT) for commercial aircraft, and a message-processing system. Other applications are also anticipated. The methodology employs a formal hierarchical decomposition of the design, with formally stated specifications for each system function and formal assertions about each desired property. This report describes the methodology, which is considered to be particularly appropriate for the task of developing the certifiable security kernels for Multics and the Secure Front-End Processor (SFEP).

The basic design approach is to isolate all nondiscretionary (i.e., mandatory) security requirements into a kernel, roughly corresponding to a stripped-down Multics Ring 0. Preliminary specifications for the kernel functions are found in Stern [76]. Bell and LaPadula [74] have precisely formulated the desired security properties for the Multics security kernel specifications. In order to support the proof of these properties, SRI has reformulated the Bell and LaPadula model in terms of the primitives of the methodology. This reformulation is summarized here, and provides the basis for proof of the correspondence between these properties and the specifications for the Multics kernel.

INTRODUCTION

Two concepts are basic here, namely SECURITY and INTEGRITY. Intuitively speaking, security deals with the SENSITIVITY of information, and is intended to prevent unauthorized READING of information (i.e., a COMPROMISE of sensitive information). Integrity, on the other hand, deals with the trustworthiness of information, and is intended to prevent unauthorized WRITING (or overwriting) of information (i.e., a CONTAMINATION of trustworthy information).

In order to be applied to a computer system, these concepts must be changed somewhat. A computer system, instead of having people reading and writing information, has only INFORMATION TRANSFER between the various information repositories in the system, including the input-output devices. Thus, we assume that each information repository in a computer system has both a security level and an integrity level, and that only people who are cleared to the appropriate levels can have access to the information in a given repository. Then, security is concerned with preventing the flow of information from a repository at a given security level to one at a lower security level. Similarly, integrity is concerned with preventing the flow of information from a repository at a given integrity level to one at a higher integrity level. Thus the two concepts are duals. In our formal proofs, we define information repositories and information flow, and assign a security level and an integrity level to each such repository.

We have developed a language for writing specifications and assertions in accordance with the methodology. This language is called SPECIAL (SPECIfication and Assertion Language) (see Robinson et al. [76], Roubine et al. [76]). In addition, we have developed on-line tools to support the use of this language. These tools are intended to simplify the overall development and proof effort. They contribute to the design by providing an on-line editable form for specifications, with automated checks of syntactic consistency. These tools also contribute to the correspondence proofs of the security of the design. Additional tools have been outlined that will make the correspondence proofs almost completely automatable.

We have also developed and are continuing to develop tools for stating and proving semantic properties of programs. These tools are compatible with the tools mentioned above to support specifications and correspondence proofs. As more of these verification tools become available, semi-automatic proofs of implementation correctness will become more feasible.

This report is organized as follows. The methodology is summarized, first with respect to design and implementation, then with respect to verification. The desired properties of the Bell and LaPadula model are then reviewed. Properties of SPECIAL are

summarized, and the desired security and integrity properties of the model are explicitly reformulated using the concepts of SPECIAL. Following a brief overview of some relevant design issues, the correspondence proofs between the reformulation of the model and the specifications for the visible interface are discussed. These concepts are also seen to apply to other properties of systems. Tools to support the verification effort are also discussed, as are several implementation considerations. Detailed examples of proofs for the specifications of Stern [76] are given in Appendix A.

II. THE METHODOLOGY FOR DESIGN AND IMPLEMENTATION

Our methodology has been described in detail elsewhere (Robinson et al. [75], Robinson and Levitt [75], Neumann et al. [75]), and continues to evolve. The methodology separates the development of a computer system or subsystem into stages corresponding to

(S0) the choice of the visible interface,

(S1) the hierarchical design,

(S2) the specification of each function at each node of the hierarchy,

(S3) the definition of mappings among the data representations at connecting nodes, and

(S4) the writing of implementation programs for the functions at each node.

These stages of design and implementation are as follows.

(S0) INTERFACE DEFINITION

In the initial stage (S0), the desired visible interface is defined. In the case of Project Guardian, this is the interface to the kernel. This "top-level" interface is then decomposed into a set of MODULES (i.e., a set of facilities), each of which manages OBJECTS of a particular type. An object is a system resource such as a segment, a directory, or a process. Each module consists of a collection of FUNCTIONS (corresponding to operations and data-structure accesses). Each function has an argument list and can be invoked by a program or directly by a user. Each function is either an O-function (Operation), which changes the state of the module to which it belongs, a V-function (Value-returning), which characterizes the state of the module, or an OV-function, which both changes the state and returns a value.

(S1) HIERARCHICAL DECOMPOSITION OF THE SYSTEM

The modules of the visible interface, together with other modules whose functions are hidden by the interface but are part of the eventual implementation, are arranged into a hierarchy of collections of modules. For descriptive simplicity, we assume here that there is only one visible interface, and so we may also assume that the hierarchy is a linear ordering of these module collections, each of which can then be referred to as a LEVEL. (For all cases considered here, there is no loss of generality in this simplified description.) The implementation of each level depends only on the next lower level. However, a module may be included in more than one level of the design, as for example the module supporting the "user" hardware instructions, which would be part of most levels of a typical operating system. The structure of the decomposition is thus explicitly declared at this stage.

(S2) MODULE SPECIFICATION

For each module, a FORMAL SPECIFICATION is developed (see Roubine et al. [76]). In this methodology, specifications are used that are similar to those suggested by Parnas [72]. However, we extend Parnas' original approach substantially, in that the specification language and the hierarchical structure have been formalized, and are supported by an on-line environment.

V-functions of a module are either PRIMITIVE (necessary for characterizing the state of the module) or DERIVED (computed from the values of other V-functions). Some V-functions are VISIBLE at the interface to a module (i.e., can be called by programs), while others are HIDDEN.

The specification of each O- or OV-function describes precisely the effect of that operation as a state change. The state change is defined by a set of EFFECTS, each of which relates values of primitive V-functions before the call on the specified O- or OV-function and values of those primitive V-functions after the return from that call. The specification of each V-function gives either the INITIAL VALUE of the function (if it is primitive), or its DERIVATION from primitive values (if it is derived). The specification for each visible function also gives EXCEPTION CONDITIONS for which a call on that function is to be rejected. Specifications are written independently of most implementation decisions concerning the module. For a well-conceived module, the specification is usually much easier to understand than its implementing program (see (S4)). Specifications are written in the language SPECIAL, discussed below. Completeness is implied by the semantics of SPECIAL, in that any primitive V-function values that are not mentioned in the specification of an O- or OV-function remain unchanged. Therefore the omission of a desired effect may result in an unfortunate specification, but will not result in an inconsistent one. Consequently, there can be no unspecified side-effects at the level of the specified function.

(S3) MAPPING FUNCTIONS

For each module above the lowest level of the design, a MAPPING FUNCTION is written that characterizes the state of the module in terms of the states of lower-level modules. A mapping function is written as a set of expansion rules, in each of which a higher-level V-function value is expanded as an expression containing lower-level V-function values. These expansion rules, called MAPPING FUNCTION EXPRESSIONS, are also written in SPECIAL. In this way assumptions are explicitly stated as to how the data structures of a module are represented in terms of lower-level modules. For example, a mapping function would relate the data structure of a user process to that of a system process, or a segment to a sequence of pages.

(S4) IMPLEMENTATION

Programs are then written to implement the visible functions of each level (except for the lowest level) in terms of those at the next lower level. Such programs are called ABSTRACT PROGRAMS since they use functions that implement abstract operations specified at lower levels of the system. Various implementation languages are possible. For present purposes, PL/I or a subset thereof may be considered adequate. Such programs may be compiled directly (with calls on lower-level programs). Optimization can them lead to more efficient programs by application of correctness preserving transformations.

Stages 0, 1, 2, 3 and 4 formalize respectively the following five conventional steps in software development: interface definition and decomposition; system modularization; specification; data representation; and coding. The results of stages 0, 1 and 2 are considered to constitute the DESIGN, while those of Stages 3 and 4 constitute the IMPLEMENTATION. In the methodology referred to here, stages 2, 3, and 4 are carried out for each level in the hierarchy.

III. THE METHODOLOGY FOR VERIFICATION

The stages of development provide the basis for the verification effort. Associated with each of these stages of design and implementation is the statement or verification of correctness properties appropriate for that stage, for each level in the hierarchical design. At Stage 0, the desired properties of the system can be explicitly formulated. One set of such properties is given by the *-property and the simple security condition of Bell and LaPadula [74], along with their duals for integrity. These properties are discussed in the next section. Other properties are also mentioned in this paper. At Stage 1, the consistency of the hierarchical structure and of the naming of functions can be demonstrated. At stage 2, the desired properties can be proven about the design (i.e., about the

specifications of the visible interface), independent of subsequent implementations. Proofs are based on

* syntactic properties of SPECIAL (SYNTACTIC properties -- for the purposes of this paper -- are those that are algorithmically checkable, while SEMANTIC properties often need a formal -- undecidable -- proof procedure to establish),

* syntactic properties not in SPECIAL but required of all specifications in the system under consideration,

* semantic properties of the specifications.

In addition, each module specification can be shown to be self-consistent (i.e., satisfiable) at this stage. At stage 3, it can be proved that the mapping functions are consistent with the specifications and the hierarchical decomposition. In the event of an inconsistency, it is impossible to implement the design consistently with the specifications. For example, no two distinct states at a higher level can both correspond to a single state at a lower level. Consistency of the mapping functions with the outputs of the previous stages is demonstrated similarly to self-consistency of the specifications. At stage 4, the implementation is proved to be consistent with the specifications and mapping functions resulting from the previous stages. Proofs of implementation consistency are done level by level. A given program intended to implement a visible V-, O-, or OV-function is proved correct with respect to its specifications, the specifications of the modules that implement the program, and the mapping function expressions relating these modules.

Once a verified system is obtained, it will tend to evolve with time. Changes in specifications and in implementations require corresponding reverification. However, reverification is required only where changes in specifications and implementations have affected the validity of the earlier verification. The staged application of this methodology and the formally defined modular decomposition can considerably simplify the reverification effort.

On the basis of its applications to date, the staged development appears to give -- from one stage to the next -- successively greater confidence in the resulting systems, first in terms of the suitability of the design and then in terms of the correctness of its implementation. Subtle design bugs have been discovered relatively easily in attempting proofs. Significant savings in development costs can result from detection of inherent insecurity in the design or implementation as early as possible. Thus there is great desirability for automated tools wherever possible.

IV. THE BASIC MULTILEVEL SECURITY MODEL

The security model of Bell and LaPadula [74] is considered next. For security, each object (i.e., operating system resource such as a segment or process) being written into or read from has a classification level and a category set, collectively referred to as the OBJECT SECURITY LEVEL. Also, each user has a clearance level and a category set, collectively referred to as the USER SECURITY LEVEL. Clearance and classification levels are linearly ordered, e.g., TOP SECRET, SECRET, etc., while category sets are partially ordered. One security level is AT LEAST that of another if and only if its classification or clearance level is at least that of the other and its category set contains the category set of the other. Similarly, for integrity, each object or user has its own integrity level, and partial ordering is defined as for security levels. The ordered pair consisting of the security level and the integrity level is called the ACCESS LEVEL. (To avoid confusion, each such level is always identified by an adjective. The term "level" used by itself refers to a collection of modules of the hierarchical design.)

The Bell and LaPadula model is expressed as follows.

SECURITY CONDITIONS:

The *-property for security: Writing is permitted only into an object with AT LEAST the user's security level. That is, there is no writing downward in security level.

The simple security condition: Reading is permitted only from an object with AT MOST the user's security level. That is, there is no reading upward in security level.

Note that writing up is not considered to be insecure, but is nevertheless often undesirable. For example, overwriting existing information at a higher security level could be very damaging. Thus writing up will also be forbidden in many cases.

The desired integrity conditions are formally the duals of these two security conditions, as follows.

INTEGRITY CONDITIONS:

Writing is permitted only into an object with AT MOST the user's integrity level. That is, there is no writing upward in integrity level.

Reading is permitted only from an object with AT LEAST the user's integrity level. That is, there is no reading downward in integrity level.

In order to prove that these four properties hold with respect to formal specifications, it is desirable to restate them in terms

of the specification and assertion language, SPECIAL, which is discussed next. We then give the reformulation of the desired security properties in the form to be proven.

V. SPECIFICATION LANGUAGE PROPERTIES RELATED TO SECURITY

SPECIAL is a formal nonprocedural specification language. It permits each function of a module to be specified independently of its implementation, so that properties of the design may be stated (in SPECIAL) and proved, independent of any implementation.

Using the language SPECIAL, the effects of an O- or OV-function of a module of some level are defined by the new values of the primitive V-functions of the level, related to the old values of those V-functions, to the arguments to the specified function, and to the parameters of the modules of the given level. (A PARAMETER of a module is a symbolic constant that is fixed for each particular instance of that module, as for example the maximum size of a segment.) Similarly, the value of a derived V-function or an exception condition is defined in terms of the values of the primitive V-functions of the level, the arguments of the specified derived V-function, and parameters of the level. The initial values of primitive V-functions are defined in terms of the module parameters and the arguments of the function. The following definitions are useful.

> A primitive V-function value is MODIFIED by the specified function if and only if it appears as a new value in an effect.
>
> A primitive V-function value is CITED by the specified function if and only if it appears as an old value in either an effect, an exception, or a derivation.
>
> SPECIAL requires that all V-function values cited or modified in any module specifications must be V-functions of the design level to which the given module belongs.
> A WRITE REFERENCE is a modified V-function value, or the value returned by a visible V-function or an OV-function, or the value of an exception condition.
>
> A READ REFERENCE is a cited V-function value, or a parameter of the level of the specified function, or an argument to the call on that function.
>
> A write reference is DEPENDENT on a read reference in a specification if and only if there exist two different legitimate values for the read reference that would cause the write reference to assume correspondingly different values.

It should be noted that exception conditions are included in the definition of a write reference because the presence or absence of an exception condition can itself result in information transfer. The occurrence of an exception condition can be viewed as a status value that is returned for each function call.

VI. REFORMULATION OF THE SECURITY PROPERTIES

The access properties given in Section IV are now reformulated. In the specifications of each function of the visible interface, each primitive V-function has an access level associated with it. Similarly, each process able to call that interface has an access level. The arguments of all visible functions are without loss of generality assumed to be at the same level as the caller (i,e., the process invoking the function), since they are supplied by the caller. (If they originated at a lower level, they could have been copied upward.) The desired security conditions may then be expressed in terms of SPECIAL as follows.

SECURITY CONDITIONS:

In the specification of a visible function, the security level of each write reference must be

(a) AT LEAST the security level of the caller, and

(b) AT LEAST the security level of each read reference upon which the write reference is dependent.

These properties satisfy the basic intuitive notion that there should be no flow of information downward to a lower security level. The duals for integrity are achieved by interchanging SECURITY and INTEGRITY, and interchanging AT LEAST and AT MOST, as follows.

INTEGRITY CONDITIONS:

In the specification of a visible function, the integrity level of each write reference must be

(a) AT MOST the integrity level of the caller, and

(b) AT MOST the integrity level of each read reference upon which the write reference is dependent.

These conditions must be satisfied by the specifications for the multilevel security interface for the Multics kernel. In fact the illustrative proofs of Appendix A show that the specifications can be written so that these properties can be proved automatically. We omit showing that the stated conditions imply conformance with the Bell and LaPadula model as stated here. However, the reformulation given here is slightly

different from that of Bell and LaPadula in that it permits writing up of information derived from a security level higher than that of the caller. This is not a violation of security, and permits greater flexibility in the design.

The correspondence between these properties and the specifications for a set of visible functions for some system guarantees that if the system was initially in a secure state, that it will subsequently be in a secure state after the execution of a call on any of the functions of the visible interface. It remains, however, to show that the initial state is secure. Checks on the consistency of the initial conditions can be made in a way similar to the above correspondence proofs.

In Millen [75], there are three additional conditions. These relate to the invariance of access levels of information repositories (the TRANQUILITY PROPERTY), to the prohibition from reading deleted information, and to the compulsory overwriting of deleted information before reuse (these last two properties called the RESIDUE PROPERTIES). These properties are trivially satisfied by our approach, as seen in Section VIII.

The above multilevel access properties must then hold for the specifications of every O-, OV-, and V-function visible at the interface. That is, no calls of a visible O-, OV-, or V-function are defined except those satisfying the above security and integrity conditions. Instead of returning an exception on a prohibited call, every possible call follows the security rules. The proof technique is illustrated below, following a summary of a typical design to support the access properties.

VII. SPECIFICATIONS FOR THE MULTICS KERNEL TO SUPPORT THE MULTILEVEL ACCESS PROPERTIES

The interface to the Multics kernel looks much like the existing Multics Ring 0 interface, except that security levels and integrity levels are associated with every object visible at that interface. However, much of the unnecessary portions of Ring 0 have been or will have been removed, incorporating some of the ideas of Schroeder [75].

The interface visible outside of the kernel is formed from the interfaces of various modules, for which specifications exist (Stern [76]). This interface includes the following modules. (Note that input-output is currently missing, along with the trusted subjects , of reconfiguration and administration --e.g., of reclassification)

address spaces
processes
segments
quota cells

volumes (on secondary storage)
access levels (security and integrity levels)
clock

An overview of a proposed kernel design decomposition is as follows, from highest kernel level to the lowest. The hardware consists of the main processors (Honeywell 68/80) and the secure front-end processor (SFEP).

user-visible input-output (to support the SFEP)
address spaces
(user) processes
segments
quota cells
volumes
access levels
supervisor-only segments
paging
input-output for system storage devices
system processes
interrupts and faults
resident segments
input-output hardware other than the SFEP
machine instruction set (Honeywell 68/80)
directly addressable memory (Honeywell 68/80)
clock (Honeywell 68/80)

Note that the objects of the top six levels of the decomposition all have access levels associated with them. The lower-level objects need not, although it may be convenient to give some of them security levels for purposes of proving program correctness. Note also that the SFEP has its own interface and its own decomposition into software and hardware.

VIII. CORRESPONDENCE PROOFS BETWEEN SPECIFICATIONS AND THE MULTILEVEL ACCESS PROPERTIES

We now describe the proof procedures used to demonstrate multilevel security in the Multics kernel. Proofs of the correspondence between the specifications and the reformulated multilevel access properties depend on the syntax and the semantics of the specifications. As noted above, the syntactic properties are of two kinds: those that are intrinsic in the language, and those that are extrinsic to the language but imposed on all specifications to be proved secure. As seen below, we rely on syntactic properties wherever possible, in order to simplify the proof effort.

The proof is based on the permanent association of access levels with V-function instantiations and processes (e.g., users). A V-FUNCTION INSTANTIATION is the V-function together with a specific set of argument values for the V-function. The system

description must include functions that map each V-function and process into an access level. The identity of the calling process is implicitly available as an argument to each visible function. The access level of a V-function instantiation cannot be changed, because to change the level would require changing the V-function instantiation itself. The access level of a user cannot be changed while the user is logged in. As for the additional properties mentioned above, the tranquility property is trivially satisfied (changing the access level of a V-function instantiation is equivalent to changing the instantiation itself), and the residue properties are not needed (even residues now have access levels), because of this permanent correspondence. It therefore remains to show that the multilevel access properties are satisfied by the specifications.

The above definitions of read reference and write reference are purely syntactic. However, the above definition of dependency requires semantics as well. A more restrictive syntactic definition of dependency can also be given, e.g., a write reference is dependent on a read reference in a given function if and only if both references occur in the same exception, effect, or derivation within that function.

There are several problems with a purely syntactic definition of dependency, however. Note that in the following effect, 'V(x) = V(x) + V1(x) - V1(x), the new value 'V(x) is not dependent on the old value V1(x) under the semantic definition of dependency, although it would be under the syntactic definition. However, 'V(x) is dependent on V(x) under both definitions. Note also that an effect of the form 'V(x) = V(x) refering to a V-function position of a security level lower than the security level of the caller is a violation of Security Condition (a) under the syntactic definition of dependency, but is not a violation under the semantic definition. Thus it seems that the best solution to the problem of detecting dependency is to use syntactic checks, and then to perform semantic checks on the cases that violate the syntactic criteria.

A specification that mentions, but does not uniquely define, the new value of a primitive V-function in terms of its dependent read references is said to be NONDETERMINISTIC. Nondeterminism is a potential source of information leakage, for example, if a malicious programmer chooses to signal information via some channel involving that nondeterminism. We can eliminate nondeterminism from a specification by insisting that each effect and derivation be written in a canonical form (e.g., <write reference> = <expression containing only read references>), with special semantic rules applying to quantification.

Arbitrary security policies may also be imposed on top of such mechanisms. An example is provided by the access control list of Multics. Although these are not a part of the presently designed security kernel, the proof approach is also applicable to such

policies.

IX. TOOLS TO SUPPORT THE DESIGN AND THE CORRESPONDENCE PROOFS

We have developed an on-line environment to support the first four stages of the methodology, i.e., the interface definition, the hierarchical decomposition, the specifications, and the mapping functions. It is also useful in performing the syntactic checks needed in the proofs of correspondence between the desired properties and the specifications. The design of this environment is open-ended, and is expected to be extended to support implementations and proofs of implementations.

The environment currently runs on TENEX. The necessary translation routines have been written to convert the INTERLISP programs on TENEX to MACLISP for Multics, so that the environment could rather easily be made to run on Multics -- although the error handling mechanism embedded in INTERLISP is different from that in MACLISP. The environment is directly applicable to the Multics security kernel. The environment currently exists in three parts, as follows.

(P1) The HIERARCHY MANAGER, which permits the establishment of a hierarchy of collections of modules, and which is responsible for maintaining the design structure.

(P2) The SPECIFICATION ANALYZER, which determines if each module specification is syntactically correct. This part includes type checking.

(P3) The MAPPING FUNCTION ANALYZER, which determines if the mapping function expressions are syntactically correct and syntactically consistent with the specifications of the modules involved.

In addition to these existing tools, a fourth tool is desirable to prove those cases involving semantic dependencies in the correspondence proofs.

(P4) The MODEL CONSISTENCY CHECKER, which performs the syntactic checks for correspondence proofs that are not a part of the specification language syntax checking, which performs simple semantic checks, and which also generates logical formulae whose validity is equivalent to the satisfaction of the more complicated semantic conditions for consistency with the model.

Based on experience to date, the generation of the logical formulae is straightforward. These conditions can be proved by hand or with machine assistance. Doing proofs on-line will be helpful in eliminating human error from the proof process. Essentially all of the correspondence proof effort can be mechanized by these tools. That is, all but a few special cases

can be treated automatically. The remaining cases, once identified, can be characterized, and most of those can then be treated automatically from then on by generalizing the special cases.

X. IMPLEMENTATION CONSIDERATIONS

The choice of a programming language for use with the methodology is not critical with respect to merely obtaining an implementation. PL/I is suitable for the task, although it is desirable to constrain the language somewhat to increase the correctness of the resulting code. However, the choice of programming language strongly influences the provability of the resulting programs. To support proofs of program correctness, the language should be well structured and should provide considerable intrinsic security, e.g., via strong type checking and restrictive scope rules. It must relate well to the methodology. It should simplify the task of program verification. It should include some of the desirable features of EUCLID, ALPHARD, SIMULA and CLU (such as protection and data abstraction).

The problem of implementing a module that is shared by concurrent processes is important in a general way, as well as with respect to security. The SRI methodology includes a model of concurrent computation with which it is possible to state and prove that a shared implementation is correct. In addition, special synchronization conditions have been derived under which a set of correct stand-alone programs may be automatically modified so that together they constitute a correct concurrent implementation. Thus programs can be verified in isolation. If the required synchronization conditions are satisfied, correctness in the real operating environment is immediately assured, given correct hardware operation.

Thus the correctness of implementation depends on the correctness of the specifications, the consistency of the implementation with the specifications without regard to concurrency, and the correctness of the synchronization conditions.

XI. TOOLS TO SUPPORT IMPLEMENTATION

In addition to the tools outlined above to support the design and the correspondence proofs, the following tools are also under development at SRI to support implementation and program verification.

> (P5) The PROGRAM HANDLER, which determines if each program is syntactically correct, and which can also perform simple semantic checks on the programs, such as those for the set of synchronization conditions noted above.

(P6) The DEVELOPMENT DATA-BASE MANAGER, which maintains a data base of the specifications, programs, and proofs in (P1), (P2), (P3), (P4), and (P5), keeping track of which modules are specified, mapped, implemented, and verified.

XII. CONCLUSIONS

The methodology discussed here has been applied to the specification of several secure systems, and proofs of security properties of the specifications have been carried out, partly manually and partly with the help of on-line tools. The correspondence proofs are seen to be quite simple, since they are aided by 1) the syntax of the specification language and its specification analyzer, 2) the abstraction afforded by the specifications, and 3) the simplicity of the model. It is expected that essentially all of the correspondence proof effort can be automated by the tools outlined here.

As a side note for the future, we have also made considerable progress in proving the consistency of implementations and specifications. The synchronization conditions are simple to state and prove, and quite powerful. Thus the correspondence between specification and implementation can be highly credible, even in the absence of complete correctness proofs.

REFERENCES

Bell and LaPadula [74] D. E. Bell and L. J. LaPadula, Secure Computer Systems: Mathematical Foundations and Model, MITRE Corp., Bedford MA (September 1974).

Lipner [75] S. B. Lipner, A Comment on the Confinement Problem, PROC. FIFTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, ACM SIGOPS REVIEW, vol. 9, no. 5, pp. 192 - 196 (19-21 November 1975).

Millen [75] J. K. Millen, Security Kernel Validation in Practice, CACM vol. 19 no. 5, pp. 243-250 (May 1976).

Neumann [74] P. G. Neumann, Toward a methodology for designing large systems and verifying their properties, 4. Jahrestagung, Gesellschaft fur Informatik, Berlin, October 9-12, 1974, in LECTURE NOTES IN COMPUTER SCIENCE, vol. 26, Springer Verlag, Berlin, 1974, pp. 52-67.

Neumann et al. [75] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena, SRI Final Report, Project 2581, 13 June 1975.

Parnas [72] D. L. Parnas, A Technique for Software Module Specification with Examples, CACM vol. 15 no. 5, pp. 330-336 (May 1972).

Robinson and Levitt [75] L. Robinson and K. N. Levitt, Proof Techniques for Hierarchically Structured Programs, SRI Report (January 1975). To appear in a future ACM publication.

Robinson et al. [75] L. Robinson, K. N. Levitt, P. G. Neumann, and A. K. Saxena, On Attaining Reliable Software for a Secure Operating System, PROC. INTERNATIONAL CONF. ON RELIABLE SOFTWARE, SIGPLAN NOTICES, vol. 10 no. 6, pp. 267-284 (June 1975). A revised and extended version is being published under the title, "A Formal Methodology for the Design of Operating System Software," in R. T. Yeh (ed.), CURRENT TRENDS IN PROGRAMMING METHODOLOGY, vol. 1, Prentice-Hall (1976).

Robinson [76], L. Robinson, Specification Techniques, Proc. 13th Design Automation Conference, IEEE cat. 76-CH1098-3C, pp. 470 - 478 (28-30 June 1976).

Roubine and Robinson [76] O. Roubine and L. Robinson, SPECIAL (SPECIfication and Assertion Language): Reference Manual, SRI Technical Report CSG-45 (August 1976), also issued as Honeywell TCL No. 23.

Schroeder [75] M. D. Schroeder, Engineering a Security Kernel for Multics, PROC. FIFTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, ACM SIGOPS REVIEW, vol. 9 no. 5, pp. 25-32 (19-21 November 1975).

Stern [76] J. Stern, Multics Security Kernel -- Top-Level Specification, Project Guardian Technical Report, Honeywell, November 1976.

Wensley et al. [76] J. H. Wensley, M. W. Green, K. N. Levitt, R. E. Shostak, The Design, Analysis, and Verification of the SIFT Fault-Tolerant System, SECOND INT. CONF. ON SOFTWARE ENGINEERING, San Francisco CA (13-15 October 1976).

MULTICS SECURITY KERNEL CERTIFICATION PLAN

APPENDIX A

METHOD FOR PROVING MULTILEVEL SECURITY

R. Feiertag

ABSTRACT

This paper presents a formal model of multilevel security. This new model is attractive because it has a simple intuitive interpretation and can be directly applied to proving the multilevel security of systems whose designs are specified in the specification language SPECIAL. The multilevel security model developed by Bell and LaPadula can be derived as a special case of the general model described below and the security properties (i.e., the simple security property and the *-property) of Bell and LaPadula are roughly equivalent to the strong properties (P2) below. It is shown how the model described below can be applied to the proof of multilevel security of system designs expressed in SPECIAL and an example of the proof technique is given. The possibility of performing the proofs by semiautomated means is then discussed.

MULTILEVEL SECURITY

In a multilevel secure system there is a predefined set of security levels. The security levels are composed of clearances (or classifications) and category sets, but the composition of the security levels is an unimportant detail for purposes of this discussion and will be largely ignored. What is important is that the security levels are partially ordered by the relation "less than" represented by "<". Each process in a multilevel secure system is assigned a security level. The processes may invoke functions that change the state of the system and return values. Each function instantiation (i.e., a function with a particular set of argument values) is assigned a security level. A process may only invoke those instantiations of funtions that have been assigned the security level of the process. A system is multilevel secure if and only if the behavior of a process at some given security level can be affected only by processes at a security level less than or equal to the given level. Stated in terms of functions, this says that the values returned by a function instantiation assigned some security level can be affected only by the invocation of function instantiations at lower or equal security levels. Stated in loose terms this means that information can flow only upward in the system from processes of lower security level to processes of higher security level.

FORMAL MODEL OF MULTILEVEL SECURITY

A multilevel system is defined to be the following ordered n-tuple:

$$<S, s_0, L, "<", I, K, R, N>$$

where the elements of the system can be intuitively interpreted as follows:

- S - States: the set of states of the system
- s_0 - Initial state: the initial state of the system; $s_0 \in S$
- L - Security levels: the set of security levels of the system
- "<" - Security relation: a relation on the elements of L that partially orders the elements of L
- I - Visible function instantiations: the set of specifications of all the visible functions and operations; if a function or operation requires arguments then the function specification along with each possible set of arguments is a separate element of I
- K - Function instantiation level: a function from I to L giving the security level associated with each visible function instantiation; K;I->L
- R - Results: the set of possible values of the visible function instantiations
- N - Interpreter: a function from IXS to RXS that defines how a given visible function instantiation invoked when the system is in given state produces a new state and a result; N:IXS->RXS.

The precise interpretation of this model for the Multics specification will be given below.

In order to define the model of multilevel security, it is useful to define the following functions:

- F(t) - the value of the function F is the first element of the ordered n-tuple t
- Z(t) - the value of the function Z is the last element of the ordered n-tuple t
- B(t) - the value of the function B is the ordered n-tuple t with the last element removed

C(t,e) the value of the function C is the ordered n-tuple t with the element e added at the end.

The following parts of the model can now be defined:

T The set of all finite ordered n-tuples of visible function instantiations or, in other words, all possible sequences of operations

$T = I^*$

M The state resulting from the given sequence of operations starting at some given state

M;SXT->S
M(s,t) = Z(N(Z(t), M(s, B(t))))

E The sequence of operations that results when all the operations whose level is not less than the given level are removed from the given sequence of operations.

E;TXL->T
E(t,1) = (K(Z(t))<1 V K(Z(t))=1) => C(E(B(t),1), Z(t))
V ~(K(Z(t))<1 V K(Z(t))=1) => E(B(t),1)

Multilevel security can now be defined as follows:

$$(\forall i<I, s<S, t_1, t_2<T)$$

$$E(t_1, K(i)) = E(t_2, K(i)) \qquad (P_1)$$

$$\Rightarrow F(N(i, M(s, t_1))) = F(N(i, M(s, t_2)))$$

This says that if two sequences of operations are each applied to a system in the same state and if these sequences differ only in operations whose level is not less than or equal to some level, then any operation of that level that is invoked immediately following the two sequences will return the same result. In other words, the operations whose level is not less than or equal to this level cannot effect results visible to the level.

STRONG MULTILEVEL SECURITY PROPERTIES

Unfortunately, it is difficult to prove that any specification meets this definition because any direct proof would require some induction on all possible sequences of function instantiations. The number of such sequences is generally very large. For this reason the following slightly more restrictive set of properties is more useful for proof because it does not involve sequences of function instantiations.

It is first necessary to introduct the notion of a partial state. There is a partial state set S^1 for each security level 1 of the system. The cross product of all the partial state sets $(\underset{\forall 1<L}{X} S^1)$ is isomorphic to the set of states (S). Therefore, each state $s<S$ can be represented by the ordered n-tuple consisting of one element s^1 from each of the partial state sets S^1.

Intuitively, one can think of a partial state set as all the state variables assigned a given security level and a partial state set as one set of values for these state variables.

The following useful functions can now be defined:

$Q_1 : s \rightarrow S^1$	has as its value the partial state of each $s<S$ for the level 1.
$Q^1 : s \rightarrow \underset{\forall k<L \mid k<=1}{X} S^k$	has as its value the partial state of each $s<S$ for all levels less than or equal to 1
$D_1 : s \rightarrow \underset{\forall k<L \mid \sim(1<=k)}{X} S^k$	has as its value the partial state of each $s<S$ for all levels not greater than or equal to 1

It is now possible to define three new security properties whose conjunction is stronger than P1 above:

$$(\forall i < I)\quad (\exists j)(\forall s < S)\ F(N(i,s)) = j(Q_{K(i)}(s)) \qquad \text{(P2a)}$$

$$(\forall i < I, 1 < L)\ (\exists j)(\forall s < S)\ Q_1(Z(N(i,s))) = j(Q_1(s)) \qquad \text{(P2b)}$$

$$(\forall i < I, s < S)\ D_{K(i)}(s) = D_{K(i)}(Z(N(i,s))) \qquad \text{(P2c)}$$

The first property (P2a) states that the result of a function instantiation at some level can be dependent only upon state variables of a lower or equal level. The second property (P2b) states that the value assumed by a state variable at some level due to the action of some function invocation can be dependent only upon state variables at a lower or equal level. The third property (P2c) states that a function invocation at some level can only change the values of state variables a greater or equal level.

PROOF OF STRONG PROPERTIES

The following is an outline of the proof that the strong multilevel security properties (P2a, P2b, P2c) imply the general multilevel security property (P1); in other words that

$$P2a \ \&\ P2b \ \&\ P2c \Rightarrow P1 \qquad \text{(T1)}$$

Using P2a in the last part of P1 yields:

$$(\forall i < I, t_1, t_2 < T)\quad (\exists j)(\forall s < S)$$

$$1(Q_{K(i)}(M(s,t_1))) = j(Q_{K(i)}(M(s,t_2)))$$

$$\Rightarrow F(N(i,M(s,t_1))) = F(N(i,M(s,t_2)))$$

and by eliminating the function j, the formula to be proven becomes:

P2a & P2b & P2c

=>

$(\forall i \leq I, s \leq S, t_1, t_2 \leq T)$ (F1)

$E(t_1, K(i)) = E(t_2, K(i))$

$\Rightarrow Q^{K(i)}(M(s,t_1)) = Q^{K(i)}(M(s,t_2))$

Now consider the cases in P1 when $E(t_1, K(i)) = E(t_2, K(i))$ is false. In these cases the theorem T1 is trivially true. Next consider the cases where $E(t_1, K(i)) = E(t_2, K(i))$ is true. These cases require an inductive proof. The induction will be over the length of the reduced sequence E(t,K(i)). Since only the cases where the reduced form of the two strings t_1 and t_2 are equal are being considered, it is known that the lengths of the two reduced strings $E(t_1, K(i))$ and $E(t_2, K(i))$ will be equal.

The basis of the induction is a reduced length of 0. In this case the sequences t_1 and t_2 can contain only function instantiations whose level is not less than or equal to K(i). From property P2c one can observe that a function instantiation whose level is not less than or equal to K(I) cannot change the partial state of the system at levels less than or equal to K(i). Therefore, the partial state at levels less than or equal to K(i) must remain the same for sequences whose reduced length is 0. For these sequences:

$$Q^{K(i)}(M(s,t)) = Q^{K(i)}(s)$$

and therefore, F1 is true.

For the purpose of accomplishing the inductive step in the proof, define the function $G_1 : T \to T$ to map a sequence of function instantiations onto the beginning of that same sequence up to but not including the last function instantiation whose level is less than or equal to 1. Also define the function $H_1 : T \to I$ to map a sequence of function instantiations onto the last function

instantiation in the sequence whose level is less than or equal to l. If a sequence t has reduced length n with respect to some level l then the sequence $G_l(t)$ has reduced length n-1 with respect to l. The induction hypothesis states that $Q^{K(i)}(M(s,G^{K(i)}(t_1)))=Q^{K(i)}(M(s,G^{K(i)}(t_2)))$ for any two sequences, t_1 and t_2, whose reduced sequences are equal. Now it is necessary to show that the last parts of sequences t_1 and t_2 make identical changes to the partial states for levels less than or equal to K(i). If $H^{K(i)}(t_1)$ is not equal to $H^{K(i)}(t_2)$ then $E(t_1,K(i))=E(t_2,K(i))$ is false and F1 is trivially true. Recall that property P2b states that any partial state at some level that results from the invocation of a function instantiation must be a function of partial states with lower or equal level. Therefore, the partial states with level less than or equal to K(i) resulting from the invocation of $H^{K(i)}(t_1)$ and $H^{K(i)}(t_2)$ must be functions of $Q^{K(i)}(M(s,G^{K(i)}(t_1)))$ and $Q^{K(i)}(M(s,G^{K(i)}(t_2)))$ respectively. If $H^{K(i)}(t_1)$ is equal to $H^{K(i)}(t_2)$ and since $Q^{K(i)}(M(s,G^{K(i)}(t_1))) = Q^{K(i)}(M(s,G^{K(i)}(t_2)))$ from the induction hypothesis then the partial states resulting from the invocations of $H^{K(i)}(t_1)$ and $H^{K(i)}(t_2)$ must be identical. All that can be left in the sequences t_1 and t_2 after the last function instantiation whose level is less than or equal to K(i) are obviously function instantiations whose level is not less than or equal to K(i). From P2c it is known that such function instantiations cannot change partial states with levels less than or equal to K(i). This completes the outside of the proof.

INTERPRETATION OF THE MODEL

In order to apply the security properties defined above to a particular system design, it is necessary to relate the elements of the model of a multilevel secure system to the specification language and to the particular system. Recall that the model is the following n-tuple:

$$<S, s_0, L, "<", I, K, R, N>$$

instantiation in the sequence whose level is less than or equal to l. If a sequence t has reduced length n with respect to some level l then the sequence $G_l(t)$ has reduced length n-1 with respect to l. The induction hypothesis states that $Q^{K(i)}(M(s,G_{K(i)}(t_1)))=Q^{K(i)}(M(s,G_{K(i)}(t_2)))$ for any two sequences, t_1 and t_2, whose reduced sequences are equal. Now it is necessary to show that the last parts of sequences t_1 and t_2 make identical changes to the partial states for levels less than or equal to K(i). If $H_{K(i)}(t_1)$ is not equal to $H_{K(i)}(t_2)$ then $E(t_1,K(i))=E(t_2,K(i))$ is false and F1 is trivially true. Recall that property P2b states that any partial state at some level that results from the invocation of a function instantiation must be a function of partial states with lower or equal level. Therefore, the partial states with level less than or equal to K(i) resulting from the invocation of $H_{K(i)}(t_1)$ and $H_{K(i)}(t_2)$ must be functions of $Q^{K(i)}(M(s,G_{K(i)}(t_1)))$ and $Q^{K(i)}(M(s,G_{K(i)}(t_2)))$ respectively. If $H_{K(i)}(t_1)$ is equal to $H_{K(i)}(t_2)$ and since $Q^{K(i)}(M(s,G_{K(i)}(t_1))) = Q^{K(i)}(M(s,G_{K(i)}(t_2)))$ from the induction hypothesis then the partial states resulting from the invocations of $H_{K(i)}(t_1)$ and $H_{K(i)}(t_2)$ must be identical. All that can be left in the sequences t_1 and t_2 after the last function instantiation whose level is less than or equal to K(i) are obviously function instantiations whose level is not less than or equal to K(i). From P2c it is known that such function instantiations cannot change partial states with levels less than or equal to K(i). This completes the outside of the proof.

INTERPRETATION OF THE MODEL

In order to apply the security properties defined above to a particular system design, it is necessary to relate the elements of the model of a multilevel secure system to the specification language and to the particular system. Recall that the model is the following n-tuple:

$$<S, s_0, L, "<", I, K, R, N>$$

The elements of the model can be interpreted as follows for the Multics specification:

S - States: all possible collective values of all the primitive V-functions of the specification; each state can be represented by a particular set of values that the primitive V-functions can assume.

s_0 - Initial state: the initial values of all the primitive V-functions as given in the specifications.

L - Security levels: each security level is defined by two values, the clearance and the category set; the clearances are totally ordered.

< - Security relation: the security relation is a partial ordering on the security levels; a security level is less than (<) another security level if the clearance of the security level is less than the clearance of the other security level and the category set of the security level is a subset of the category set of the other security level.

I - Visible function instantiations: each visible function of the specifications together with a set of possible argument values to that function is a visible function instantiation.

K - Function instantiation level: this is the level of the visible function instantiation and must be defined for each visible function instantiation.

R - Results: a result is the return value of a visible V- and OV-function invocation and the number of the first exception, if any, in a visible function invocation whose value is true; i.e. a result are the visible effects of the visible functions.

N - Interpreter: the semantics of the specification language.

The partial states S^1 are represented by subdividing the primitive V-function instantiations (i.e. primitive V-functions together with a particular set of argument values) into disjoint sets, one set for each security level. The partial state is determined by the value of the primitive V-function instantiations that are members of the partial state set.

STRONG SECURITY PROPERTIES IN TERMS OF SPECIFICATION LANGUAGE

The purpose of this section is to state the strong security properties P2a, P2b, and P2c in terms of constructs of the specification language. In order to formally relate the strong security properties as given above in terms of the formal model to the specification language it is necessary to have a formal description of the semantics of the specification language. Since such a formal description of the language has not been completed, this section will discuss the strong security properties in an informal manner. An English language description of SPECIAL is given in the SPECIAL Reference Manual. The following definitions will be useful in the discussion:

*A primitive V-function instantiation is said to be modified by a particular visible function instantiation iff the primitive V-function instantiation appears as a new (quoted) value in the effects section of the specification of the visible function and the value of the primitive V-function instantiation could be changed by invoking the visible function instantiation.

*A primitive V-function instantiation is said to be cited by a particular visible function instantiation iff the primitive V-function instantiation appears as an old (unquoted) value in the specification of the visible function.

*A write reference in a visible function instantiation is a primitive V-function instantiation, the return value of a V- or OV-function, or the exceptions.

*A read reference in a function instantiation is a cited primitive V-function instantiation.

*The value of a read reference is legitimate iff it can be assumed by the cited primitive V-function instantiation after some sequence of O- or OV- functions applied to the system in its initial state.

*The value of a read reference is type legitimate iff it is of the type of the cited primitive V-function.

*A write reference is dependent upon a read reference with respect to a particular function instantiation iff there exists two different legitimate values for the read reference that would cause the write reference to assume correspondingly different values as the result of the invocation of the function instantiation.

A slightly stronger from of the definition of dependency can be obtained by substituting "type legitimate" for "legitimate". It is easier to determine the type legitimate values of a read reference than it is to determine the legitimate values since type legitimacy is a property of the language whereas legitimacy is a property of a particular set of specifications. It is, therefore, easier to identify dependencies if the type legitimate version of the definition is used; however, for the purposes of this discussion either version of the definition of dependent suffices.

Given the above definitions it is possible to easily state the strong security properties in terms of the specification language. Note first that the above definition of dependence simply defines a functional relationship, i.e., if a write reference is dependent upon a read reference then the value of the write reference is simply a function of the value of the read reference. Recall that property P2a states that the result of the invocation of a function instantiation of some level is a function of (i.e., is dependent upon) the values of the state variables (i.e., the primitive V-function instantiations) of lower or equal levels. The results are the return values of V- and OV-functions and the exception conditions of all visible functions. Therefore, property P2a can be restated as:

P2a The return value of a V- or OV-function and the exceptions of a visible function instantiation can be dependent, with respect to that visible function instantiation, only upon read references of lower or equal level.

Property P2b states that the values assumed by a state variable (i.e., modified primitive V-function instantiation) at some level can be dependent, with respect to a visible function instantiation, only upon state variables (i.e., cited primitive V-function instantiations) at a lower or equal level. Restated this is:

P2b The value assumed by a modified primitive V-function instantiation at some level can be dependent, with respect to a visible function instantiation, only upon read references at a lower or equal level.

The similarity in the restatements of properties P2a and P2b and the fact that the return value, exceptions, and modified primitive V-function instantiations of a visible function are simply the write references of the function allows the following combination of the statements of the two properties into:

P2a,b For each visible function instantiation, the security level of each write reference must be at least the security level of each read reference upon which the write reference is dependent.

Property P2c states that the invocation of a function instantiation at some level can change only the values of state variables (i.e., modified primitive V-functions) at greater or equal levels. If the return value and the exceptions are defined to be at the level of the function instantiation of which they are a part then this property can be restated as:

P2c For each visible function instantiation, the security level of each write reference must be at least the security level of the function instantiation.

Combining this statement and P2a,b above gives a general restatement of the strong security properties in terms of SPECIAL:

P3 For each visible function instantiation, the security level of each write reference must be at least the security level of:

(a) the function instantiation, and

(b) each read reference upon which the write reference is dependent.

Given a formal description of the semantics of SPECIAL, property P3 can be formally stated and the logical statement P3 => P2 can be rigorously proven true.

DETERMINING DEPENDENCIES

This section discusses means for identifying dependencies. The objective is to find some simple algorithm for identifying dependencies. Unfortunately, determining if some write reference is dependent upon some read reference is, in general, undecidable. The approach taken here is to identify potential dependencies. If the set of all write references of a specification is W and the set of all read references is R, then the dependency relation DR is a subset of WXR and the potential dependency relation PDR is a subset of WXR and a superset of DR. If property P3 can be proven for potential dependencies rather than for dependencies, then clearly P3 must be true for dependencies. Property P3 for potential dependencies rather than dependencies will be termed P4. The problem then becomes to identify the set of potential dependencies and show that all dependencies are included in this set. However, the cardinality of the set of potential dependencies must be kept as small as possble to make the proof of P4 tractable.

In order to simplify the following discussion, it will be assumed that the specifications are in expanded form. An expanded specification is one in which the substitutions resulting from DEFINITIONS, EXCEPTION_OF, and EFFECTS_OF expressions have been performed. These substitutions are straightforward. In and expanded specification all read and write references relevant to a visible function instantiation will be explicitly present in the body of that visible function's specification. Specifications may still be written in unexpanded terms of expanded specifications.

There are certain types of expressions that are legal in SPECIAL, but make it very difficult to determine if dependencies or potential dependencies exist. To eliminate the necessity of dealing with such expressions a canonical form for specifications is introduced. The canonical form is a restriction of SPECIAL. In the canonical form, the grammar of SPECIAL is modified and augmented as follows. An <expression> in the body of a function specification cannot contain the symbol which is the identifier for the return value of the function. The definition of <call> is modified to be:

```
<call> ::= <symbol> '(' [<expression> ( ',' <expression>)*] ')'
```

The purpose of these two changes is to eliminate the possibity of a write reference in an<expression>. A <write reference> is either a quoted V-function or the identifier of the return value for visible function in which the <write reference> occurs. The following definitions are added (note that in the TYPECASE alternative of <canonical expression> below that <symbol> must not be the identifier of the return value):

```
<canonical expression>
  ::= <write reference> '=' <expression>
      | <canonical expression> AND <canonical expression>
      | <expression> '=>' <canonical expression>
      | (FORALL | EXISTS) <qualif\declarationlist>
        ":" <canonical expression>
      | IF <expression> THEN <canonical expression>
        ELSE <canonical expression>
      | LET <qualification> ( ';' <qualification> )*
        IN <canonical expression>
      | TYPECASE <symbol> OF
        ( <canonical case> ';' )+ END

<canonical case>
  ::= <typespecification> ':' <canonical expression>
```

and finally the definition of <effects> is changed to:

```
<effects> ::= EFFECTS ( <canonical expression> ';' )+
```

The purpose of the canonical form is to restrict how write references can occur in specifications. This canonical form was arrived at through experience with writing specifications and

attempts to prove the multilevel security of specifications. Our experience shows no specifications that do not fit into this canonical form.

In order to get some idea of how dependencies are indicated by function specifications, it is necessary to have some rough idea of the semantics of a function specification. For all visible functions the semantics of exceptions can be stated as:

$$(\forall i \mid 0<i<=n)\ ((\ \underset{0<j<i}{\text{AND}}\ \tilde{}EX_j\ )\ \text{AND}\ EX_i\ ) = (EV=i) \qquad \text{(S1a)}$$

$$(\ \underset{0<i<=n}{\text{AND}}\ \tilde{}EX_i\ ) = EV=0 \qquad \text{(S1b)}$$

where EX_i is the ith exception, n is the number of exceptions, and EV is the exception value. EV is the number of the first exception whose value is true. If all the exceptions are false, then EV is 0. In an O- or OV-function the semantics of effects are:

$$(\ \underset{0<i<=n}{\text{AND}}\ \tilde{}EX_i\ ) = (\ \underset{0<j<=m}{\text{AND}}\ EF_j\ ) \qquad \text{(S2)}$$

where EX_i, n, and EV are as above and EF_i is the ith effect and m is the number of effects. Note that in an OV-function the return value is specified by the identifier given in the function header. In a V-function the semantics of the derivation is:

$$(\ \underset{0<i<=n}{\text{AND}}\ \tilde{}EX_i\ ) = (RV=DE) \qquad \text{(S3)}$$

where EX_i, n, and EV are as above and RV is the value returned by the function and DE is the derivation.

Consider now where potential dependencies can exist. As a first approximation assume that a potential dependency exists between all write references of a visible function instantiation and all of its read references. This is clearly a superset of all the dependencies that exist with respect to the function since the semantics of SPECIAL does not allow the value of primitive V-functions not appearing in the specification of a function to be changed by the function and does not allow any new values to be dependent on nonappearing primitive V-functions. Unfortunately, this rather simple identification of potential dependencies includes too many potential dependencies and it is not possible to construct useful systems that are consistent with property P4.

Consider the three types of write references separately. First consider the value of the exceptions, EV. EV is clearly potentially dependent only upon read references in the exceptions section of the function specification. In fact, in some circumstances it may be possible to prove that for some instantiations of a visible function, that a particular exception is always true. In this case EV is potentially dependent only upon read references in exceptions coming before the one that is always true for these instantiations of the visible function.

Now consider those write references that are modified primitive V-functions. Modified primitive V-functions can only occur in the effects section of an O- or OV-function. A write reference in an effect can only be potentially dependent upon read references in that same effect and read references in the exceptions. This follows from S2 above and the canonical form. If a write reference appears in a series of conjoined expressions then the write reference is not potentially dependent on read references in any of the other conjoined expressions. This follows from the definition of conjunction and the canonical form.

Finally consider write references that are return values. If the visible function is an OV-function then the rules for modified primitive V-functions apply. If the visible function is a V-function then the return value is potentially dependent upon the read references in the exceptions and in the derivation.

In summary, the rules for potential dependency are as follows:

PDR_1 The exceptions value is potentially dependent upon read references in all exceptions up to the first exception that is always true for the visible function instantiation.

PDR_2 Each modified primitive V-function in an O- or OV-function and each return value in an OV-function is potentially dependent upon read references in exceptions and all read references in the same effect as the write reference with the exception of read references in expressions conjoined with the expression containing the write reference.

PDR_3 The return value of a visible V-function is potentially dependent upon read references in the derivation and read references in exceptions.

The following provide interesting and important exceptions to the above rules:

```
FALSE => exp_a
IF FALSE THEN exp_a ELSE exp_b
IF TRUE THEN exp_b ELSE exp_a
FORALL x INSET (): exp_a
FORALL x | FALSE: exp_a
EXISTS x INSET (): exp_a
EXISTS x | FALSE: exp_a
LET x INSET () IN exp_a
LET x | FALSE IN exp_a
```

No write reference can be dependent upon any read reference in exp_a of these expressions. This is evident from the semantics of these expressions. Although it is unlikely to see expressions precisely like these in well written specifications, it is possible that such expressions effectively exist for some instantiations of visible functions. Some examples of these will be given below.

THE PROOF TECHNIQUE

Before summarizing the steps in the proof technique, one further observation is useful. Not all quoted primitive V-functions necessarily represent modified primitive V-functions and, therefore, do not necessarily represent write references. For example in an expression of the form

```
FALSE => 'pvf(args) = exp
```

the quoted primitive V-function pvf does not represent a write reference because the expression does not constrain the value of pvf(args) to change. This situation arises in all the expressions listed in the previous paragraph as exceptions to the potential dependency rules. Similarly, a quoted primitive V-function in the effects section of a visible function instantiation in which some exception is always true is never a write reference.

The proof of multilevel security of a given specification is quite straightforward. For each visible function specification it must be shown that each instantiation of that function is consistent with property P4. This can be accomplished by proving that P4 holds for all possible argument values to the function or it can be accomplished by dividing the possible sets of arguments into collectively exhaustive subsets and then proving P4 for each of the subsets. For each subset the write references must be identified and then it must be shown that for each write reference there is a modified V-function, that the level of that V-function is greater than or equal to the level of the visible function instantiation. Finally, it must be shown that for each write reference, each read reference upon which the write reference is potentially dependent has a level less than or equal to the level of the write reference.

Unfortunately, it is not always possible to determine the level of a read or write reference in a particular function instantiation by inspection of the specification. For example, an argument to some primitive V-function might be the value of some other primitive V-function. In this case it is necessary to know what values the other primitive V-function might have in order to know what the level of the read reference is. Such information may be deducible from the specification of the visible function in question (local assertion) or it may require proving some invariant of the specification of the system as a whole (global assertion). In either case it is necessary to prove P4 for all possible values that the other primitive V-function above may assume. Examples of this case are given below.

EXAMPLE

The proof of the multilevel security of a specification will be demonstrated using the specification given in Fig. 1. It is necessary to use a rather simple example in order to be able to describe the proof within a reasonable amount of space. Proofs of large specifications are rather lengthy. The security levels of the specification of Fig. 1 are given by the definition of the security_level type. The definition of the security relation (<) is given by the definition read_allowed in the specification, i.e., the security level L1 is less than or equal to the security level L2 (L <=L2) iff read_allowed(L2, L1) is true. The level of each visible function instantiation and each primitive V-function instantiation is given in Fig. 2. Note that the specification of Fig. 1 is not in expanded form because there are definitions present. However, because these definitions contain no primitive V-functions and because they succinctly express the security relation, it is more convenient to deal with this unexpanded form.

Consider first the first visible function "create_seg". The security level of all instantiations of this function (and its arguments, parameters, exception value, and return value by definition) is the value of its last argument "s1". This function has no exceptions and, therefore, the exception value cannot be dependent on the system state. Look now at the write references in the EFFECTS section. There are three quoted primitive V-functions and one return value identifier. We must consider as potential write references all those instantiations of the quoted primitive V-functions subject to the constraints of the qualification of the EXISTS statement. Using the security levels of the primitive V-function instantiations given in Fig. 2, to demonstrate property P4a we must prove that

```
for ´h_uid_used(new_uid.id, s1):
  (∀new_uid | h_uid_used(new_uid.id, s1) ∧ new_uid.1=s1)
   s1 <= s1
```

```
for ´h_seg_exists(new_uid):
  (∀new_uid | h_uid_used(new_uid.id, s1) ∧ new_uid.1=s1)
   s1 <= new_uid. 1

and for ´h_contents(new_uid, i):
  (∀new_uid | h_uid_used(new_uid.id, s1) ∧ new_uid.1=s1)
   (∀i |0<=i<size) s1 <= new_uid.1
```

These three assertions are trivially true. Now, consider the read references of the EFFECTS section. The primitive V-function h_uid_used(new_uid.id, s1) represents several read references, one for each possible value of "new_uid.id". By rule PDR2 we know that all the write references of each instantiation of "create_seg" are potentially dependent upon these read references. In order to prove P4b, it must be shown that each of these write references is at a level at least that of the read references. Fortunately, all the read references are at the level "s1", the level of the visible function instantiation, and we have already shown that all the write references are at least at this level. This completes the proof for the function "create_seg".

Consider now the visible function "write_seg". We will consider separately four different collections of instantiations of this visible function:

```
case 1:  write_allowed(s1, suid.1) = FALSE
         AND read_allowed(s1, suid.1) = FALSE

case 2:  write_allowed(s1, suid.1) = FALSE
         AND read_allowed(s1, suid.1) = TRUE

case 3:  write_allowed(s1, suid.1) = TRUE
         AND read_allowed(s1, suid.1) = FALSE

case 4:  write_allowed(s1, suid.1) = TRUE
         AND read_allowed(s1, suid.1) = TRUE
```

In case 1, the first exception of all instantiations in the case is true, and therefore, the exception value will always be 1. For these instantiations, the exception value is the only write reference, is not dependent on any read references, and is at the level of the instantiation by definition. Property P4 is, therefore, trivially true. Case 2 follows the same reasoning. In case 3, all the exceptions are always false and the exception value is always 0 and, therefore, the exception value is not dependent on any read references for these instantiations. The only quoted primitive V-function in the EFFECTS section is h_contents(suid, offset). To prove P4a we can show that the level of all instantiations of this primitive V-function is at least "s1", the level of the visible function instantiation. The level of all instantiations of h_contents (suid, offset) is suid.1 and, since we are considering only

instantiations in case 3, we know that write_allowed(s1, suid.1) is true. We wish to prove that s1 <= suid.1, and this follows directly from write_allowed(s1, suid.1) being true. In order to prove P4b we must show that the level of the instantiation of ´h_contents(suid, offset) is at least the level of the read reference that is the unquoted version of this same primitive V-function instantiation. Since the read reference and write reference in question are for the same primitive V-function instantiation, their security levels must be the same. In case 4 the exception value is dependent on instantiations of the primitive V-functions h_seg_exists(suid) and h_contents(suid, offset). Both these primitive functions have a level of "suid.1". However, we know that suid.1 <= s1 because read_allowed(s1, suid.1) is true. The reasonging for the EFFECTS section is similar to that of case 3 except that the write reference of ´h_contents(suid, offset) is now dependent upon the value of h_seg_exists(suid) as well as the previous value of h_contents(suid, offset). However, the security levels of h_contents (suid, offset) and h_seg_exists(suid) are the same and P4b is easily satisfied. This completes the proof of multilevel security for "write_seg". The arguments for "delete_seg" and "read_seg" are quite similar.

Although the sample specification is quite simple, the same proof technique can be applied to more complex systems. The added difficulty in proving more complex systems arises from the increased number of read and write references and the more complex techniques necessary to prove relationships between the security levels of these read and write references. Systems that require the use of global assertions in the proofs are even more difficult because appropriate global assertions must be determined and the validity of these global assertions must be proven. Although the proofs may be more complex, the basic technique demonstrated in the above example does not change.

AUTOMATING THE PROOFS

The proof of the simple specification of Fig. 1 is simple but quite lengthy when fully documented. Proofs of complex systems will be extremely lengthy. In general, the proof of multilevel security of a specification is many times longer than the specification. If these proofs are written manually, the probability of their correctness is very small. Unfortunately, even a small error in the design of a system can result in a large breach of security. It is, therefore, necessary that there be a high degree of confidence in the total correctness of the security proofs. Such a high degree of confidence in the correctness of the proofs cannot be effectively gained by manual generation and checking of the proofs. The necessary degree of confidence can only be gained by automatic generation or checking of proofs or some combination of automatic and manual techniques.

The proof technique described above has been designed to permit automatic generation of proof. The identification of read references, write references, and potential dependencies of the desired property P4 can be done very simply with knowledge of the syntax and a little of the semantics of SPECIAL. The proof of the relationships between the security levels of the read and write references requires some theorem proving, but the types of theorems involved are all of the same simple kind and most of them can be handled by a simplifier. Those systems that require global assertions in order to perform the proofs will probably require human assistance in deriving the global assertions, however, the proof of the global assertions can probably be automated. For a given system specification, the same theorems arise many times in proving the security of the different visible functions. Once the security of a few of the visible functions has been proven, the proofs of the remaining functions follow similar patterns. Highly efficient operation can be achieved if the automated prover is directed by a human operator for the proofs of a few of the visible functions and then uses the same techniques to automatically prove the security of the remaining functions. Also, after the automated prover has proved the security of a system specification once and is aware of the necessary global assertions, it should be able to prove the security of modified versions of the system with human assistance. The use of such a semi-automated prover is essential to having a high degree of confidence in the proofs, is within the current state of the art of automated verification, and will be more cost effective than manual proof techniques for large systems even with the high initial cost.

Most of the tools necessary for constructing a semi-automated prover for the security of specifications already exist. There exists several theorem provers and simplifiers which should be adequate for the types of theorems that will be generated. A program exists to parse specifications written in SPECIAL and to convert to a form suitable for processing. The necessary additional programs are a verification condition generator that formulates the theorems that express the desired relationships between the security levels of the read and write references and a suitable human interface. Verification condition generators and human interfaces have been written to aid in the proof of properties in several other languages and the ideas in these programs can be used to create programs suitable for proving the multilevel security of specifications.

APPLICATION TO THE MULTICS SPECIFICATIONS

The multilevel security model and the proof technique described above can be applied to the Multics specifications. However, there are two significant discrepancies between the security model and the Multics specifications. First, the Multics specifications incorporate the notion of a trusted process, i.e., a process that is not

subject to the multilevel security constraints. Trusted processes are clearly in violation of the general multilevel security property (P1) above, therefore, it is necessary to modify the model in order to allow trusted processes to exist. The modifications necessary to P1, P2, P3, and P4 are all very simple, however, they will not be described here. The modification involves adding a new predicate to the definition of a system that is true if the given visible function instantiation is subject to the multilevel security constraints. During a proof it is not necessary to prove anything about those visible function instantiations for which the predicate is true.

The second difference arises from the necessity of partitioning the primitive V-function instantiations into disjoint sets, one for each security level. These partitions are not a function of time. However, in the Multics specifications, the security levels associated with primitive V-function instantiations do change with time. For example the security level of the primitive V-function h_seg_contents(seguid, offset) is not determined until a segment with the unique identifier seguid is created. Before the segment is created the security level of this primitive V-function is undefined. Some modification of the mathematical model might be found to permit the security level of a primitive V-function to remain initially undefined, however, it would probably be simpler to predefine the security level of all unique identifiers and assign a suitable unique identifier to a newly created segment. This solution applies equally well to all dynamically created objects.

The proof of multilevel security of the Multics specifications will require global assertions. One such global assertion would be:

```
h_kst_mode(procuid, segno)[1]
   AND h_ppr_ring(procuid) <= h_kst_rb(procuid, segno)[2]
 => h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
       h_qc_al(h_seg_qc(h_kst_seguid(procuid, segno))))
```

Although this assertion seems rather formidable, it is quite easy to prove because very few O- and OV-functions modify the values of the primitive V-function instantiations involved. Also, the number of global assertions necessary to prove the multilevel security of the Multics specifications should be small.

The Multics design also incorporates the notion of integrity. Integrity is the formal dual of multilevel security and a dual model for integrity can easily be stated. Precisely the same proof techniques apply and, therefore, proof of integrity can be easily achieved together with the proof of security.

```
MODULE SEGMENTS

  TYPES

clearance:  ( INTEGER i | 0 < i AND i <= max_clearance );
category_set:
 ( VECTOR OF BOOLEAN cs | LENGTH(cs) = number_of_categories );
security_level: STRUCT(clearance security_clearance;
                       category_set security_categories);
segment_uid:  STRUCT(INTEGER id; security_level l);


  PARAMETERS

INTEGER max_clearance s( the highest clearance ),
        number_of_categories;


  DEFINITIONS

BOOLEAN read_allowed(security_level subject_sl, object_sl)
   IS subject_sl.security_clearance
       >= object_sl.security_clearance
      AND (FORALL INTEGER i | 0<i AND i <= number_of_categories:
            object_sl.security_categories[i]
            => subject_sl.security_categories[i]);
BOOLEAN write_allowed(security_level subject_sl, object_sl)
   IS read_allowed(object_sl, subject_sl);


  FUNCTIONS

VFUN h_uid_used(INTEGER unique_integer; security_level sl)
     -> BOOLEAN b;
    s( true if unique_integer has been used before at
       security level sl)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_seg_exists(segment_uid suid) -> BOOLEAN b;
    s(true if segment suid exists )
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_contents(segment_uid suid; INTEGER offset)
     -> INTEGER contents;
    s( returns contents of word at offset in segment suid )
    HIDDEN;
    INITIALLY contents = ?;
```

Fig. 1 (continued on next page)

```
OVFUN create_seg(INTEGER size) [security_level sl]
      -> segment_uid suid;
    s( create a new segment with size number of words )
    EFFECTS
     EXISTS segment_uid new_uid | h_uid_used(new_uid.id, sl)
                                  AND new_uid.1 = sl:
      ´h_uid_used(new_uid.id, sl) = TRUE
      AND suid = new_uid
      AND ´h_seg_exists(new_uid) = TRUE
      AND (FORALL INTEGER i | 0 <= i AND i < size:
           ´h_contents(new_uid, i) = 0);

OFUN delete_seg(segment_uid suid) [security_level sl];
    s( delete a segment with uid suid )
    EXCEPTIONS
     read_allowed(sl, suid.1) AND ~h_seg_exists(suid);
     ~write_allowed(sl, suid.1);
    EFFECTS
     FORALL INTEGER i: ´h_contents(suid, i) = ?;
     ´h_seg_exists(suid) = FALSE;
VFUN read_seg(segment_uid suid; INTEGER offset)
            [security_level sl] -> INTEGER contents;
    s( returns the value of the item at offset in
       segment suid )
    EXCEPTIONS
     ~read_allowed(sl, suid.1);
      ~h_seg_exists(suid);
      h_contents(suid, offset) = ?;
    DERIVATION
      h_contents(suid, offset);

OFUN write_seg(segment_uid suid; INTEGER offset;
               INTEGER contents)[security_level sl];
    s( modify the contents of item offset in segment suid )
    EXCEPTIONS
     ~write_allowed(sl, suid.1);
     read_allowed(sl, suid.1) AND ~h_seg_exists(suid);
     read_allowed(sl, suid.1) AND h_contents(suid, offset) = ?;
    EFFECTS
     h_contents(suid, offset) ~= ?
     =>  h_contents(suid, offset) = contents;

END_MODULE
```

Fig. 1 - Example specification

Function Instantiation	Security Level
(primitive V-function instantiations)	
h_uid_used(unique_identifier, s1)	s1
h_seg_exists(suid)	suid.1
h_contents(suid, offset)	suid.1
(visible function instantiations)	
create_seg(size)[s1]	s1
delete_seg(suid)[s1]	s1
read_seg(suid, offset)[s1]	s1
write_seg(suid, offset, contents)[s1]	s1

Fig. 2 - Security levels of function instantiations.

APPENDIX B

Because of funding limitations, the Air Force terminated the effort which this document describes before the effort reached its logical conclusion. The Air Force comments which were present at the time the effort was terminated are as follows:

1. The report does not provide a complete certification plan. The report adequately treats the proof of correspondence between the model and the top level specification. There is no plan presented for the remaining effort to ultimately insure the correspondence between the model and the machine code representation of the kernel.

2. The report does not present sufficient data for a management evaluation of the cost/effectiveness of the automated tools. The report does not identify the tools that will be required for the remaining stages of the verification and the cost (effort) that will be required to implement the tools.

3. On Page A-1, the purpose of the multilevel security model is not clear. It is not known whether this model is intended as an alternative (replacement) for the MITRE model or as an intermediate step in the proof of correspondence between the MITRE model and the formal specification. The purpose of the model should be identified, and its correspondence to either the DoD security policy or the MITRE model needs to be demonstrated.

4. On Page A-2 and A-3, several of the definitions on these pages are not formally complete. For example, F, Z and B are not clearly defined for a one-tuple. The recursive definitions of M and E do not have a "base" statement.

5. On Page A-5, Line 6, the substitution of P2a in P1 appears to have been made incorrectly. P2a was substituted for the first rather than the last part of P1. It should have been the last part.

MISSION
OF THE
DIRECTORATE OF COMPUTER SYSTEMS ENGINEERING

The Directorate of Computer Systems Engineering provides ESD with technical services on matters involving computer technology to help ESD system development and acquisition offices exploit computer technology through engineering application to enhance Air Force systems and to develop guidance to minimize R&D and investment costs in the application of computer technology.

The Directorate of Computer Systems Engineering also supports AFSC to insure the transfer of computer technology and information throughout the Command, including maintaining an overview of all matters pertaining to the development, acquisition, and use of computer resources in systems in all Divisions, Centers and Laboratories and providing AFSC with a corporate memory for all problems/solutions and developing recommendations for RDT&E programs and changes in management policies to insure such problems do not reoccur.